MozTrap Documentation

Release 1.0.0

Mozilla

January 18, 2013

CONTENTS

MozTrap is a test case manager.

CHAPTER

ONE

QUICKSTART

MozTrap requires Python 2.6 or 2.7 and MySQL 5.1+ with the InnoDB backend.

These steps assume that you have git, virtualenv, virtualenvwrapper, and a compilation toolchain available (with the Python and MySQL client development header files), and that you have a local MySQL server running which your shell user has permission to create databases in. See the full *Installation* documentation for details and troubleshooting.

1. `git clone --recursive git://github.com/mozilla/moztrap`
2. `cd moztrap`
3. `mkvirtualenv moztrap`
4. `bin/install-reqs`
5. `echo "CREATE DATABASE moztrap CHARACTER SET utf8" | mysql`
6. `./manage.py syncdb --migrate`
7. `./manage.py create_default_roles`
8. `./manage.py runserver`
9. Visit http://localhost:8000 in your browser.

Congratulations! If that all worked, you have a functioning instance of MozTrap for local testing, experimentation, and *development*.

Please read the *Deployment* documentation for important security and other considerations before deploying a public instance of MozTrap.

CHAPTER

TWO

CONTENTS

2.1 Installation

2.1.1 Clone the repository

First, clone the MozTrap repository.

Dependency source distribution tarballs are stored in a git submodule, so you either need to clone with the `--recursive` option, or after cloning, from the root of the clone, run:

```
git submodule init; git submodule update
```

If you want to run the latest and greatest code, the default `master` branch is what you want. If you want to run a stable release branch, switch to it now:

```
git checkout 0.8.X
```

2.1.2 Install the Python dependencies

If you want to run this project in a virtualenv to isolate it from other Python projects on your system, create the virtualenv and activate it. Then run `bin/install-reqs` to install the dependencies for this project into your Python environment.

Installing the dependencies requires pip 1.0 or higher. pip is automatically available in a virtualenv; if not using virtualenv you may need to install it yourself.

A few of MozTrap's dependencies include C code and must be compiled. These requirements are listed in `requirements/compiled.txt`. You can either compile them yourself (the default option) or use pre-compiled packages provided by your operating system vendor.

Compiling

By default, `bin/install-reqs` installs all dependencies, including several that require compilation. This requires that you have a working compilation toolchain (`apt-get install build-essential` on Ubuntu, Xcode on OS X). It also requires the Python development headers (`apt-get install python-dev` on Ubuntu) and the MySQL client development headers (`apt-get install libmysqlclient-dev` on Ubuntu).

If you are lacking the Python development headers, you will get the error `Python.h: No such file or directory`. If you are lacking the MySQL client development files, you will get an error that `mysql_config` cannot be found.

Using operating system packages

If you prefer to use pre-compiled operating system vendor packages for the compiled dependencies, you can avoid the need for the compilation toolchain and header files. In that case, you need to install MySQLdb, py-bcrypt, and coverage (the latter only if you want test coverage data) via operating system packages (`apt-get install python-mysqldb python-bcrypt python-coverage` on Ubuntu).

If using a virtualenv, you need to ensure that it is created with access to the system packages. In virtualenv versions prior to 1.7 this was the default, in recent versions use the `--system-site-packages` flag when creating your virtualenv.

Once you have the compiled requirements installed, install the rest of the requirements using `bin/install-reqs pure`; this installs only the pure-Python requirements and doesn't attempt to compile the compiled ones. Alternatively, you can skip `bin/install-reqs` entirely and use the provided *Vendor library*.

2.1.3 Create a database

You'll need a MySQL database. If you have a local MySQL server and your user has rights to create databases on it, just run this command to create the database:

```
echo "CREATE DATABASE moztrap CHARACTER SET utf8" | mysql
```

(If you are sure that UTF-8 is the default character set for your MySQL server, you can just run `mysqladmin create moztrap` instead).

If you get an error here, your shell user may not have permissions to create a MySQL database. In that case, you'll need to append `-u someuser` to the end of that command, where `someuser` is a MySQL user who does have permission to create databases (in many cases `-u root` will work). If you have to use `-u` to create the database, then before going on to step 5 you'll also need to create a `moztrap/settings/local.py` file (copy the sample provided at `moztrap/settings/local.sample.py`), and uncomment the `DATABASES` setting, changing the `USER` key to the same username you passed to `-u`.

2.1.4 Create the database tables

Run `./manage.py syncdb --migrate` to install the database tables.

2.1.5 Create the default user roles

This step is not necessary; you can create your own user roles with whatever sets of permissions you like. But to create a default set of user roles and permissions, run `./manage.py create_default_roles`.

2.1.6 Run the development server

Run `./manage.py runserver` to run the local development server. This server is a development convenience; it's inefficient and probably insecure and should not be used in production.

2.1.7 All done!

You can access MozTrap in your browser at http://localhost:8000.

For a production deployment of MozTrap, please read the *Deployment* documentation for important security and other considerations.

For notes on upgrading to a more recent MozTrap, see the *Upgrading* documentation.

2.2 Upgrading

To upgrade, simply use git to pull in the newer code from the GitHub repository and update the submodules:

```
git pull
git submodule update
```

If you are on a stable release branch (e.g. `0.8.X`) and you want to update to a newer release branch (e.g. `0.9.X`), make sure you've fetched the latest code on all branches, switch to the branch you want, and update to the correct version of the submodules for that branch:

```
git fetch
git checkout 0.9.X
git submodule update
```

2.2.1 Updating dependencies

Run `git submodule update` to get the latest version of the dependency submodules, and then `bin/install-reqs` to install them into your environment. Both of these commands are idempotent; there's no harm in running them every time, whether there have been any dependency changes or not.

If you are using the *Vendor library*, `bin/install-reqs` is not necessary, the submodule update will get the latest version of the vendored dependencies.

2.2.2 Database migrations

It's possible that the changes you pulled in may have included one or more new database migration scripts. To run any pending migrations:

```
python manage.py syncdb --migrate
```

This command is idempotent, so there's no harm in running it after every upgrade, whether it's necessary or not.

Warning: It is possible that a database migration will include the creation of a new database table. If you've commented out the `SET storage_engine=InnoDB init_command` in your `moztrap/settings/local.py` for performance reasons (see *Database performance tweak*), you should uncomment it before running any migrations, to ensure that all new tables are created as `InnoDB` tables.

2.3 Development

2.3.1 Coding Standards

Python

Testing

All tests should pass, and 100% line and branch test coverage should be maintained, at every commit (on the master branch or a release branch; temporary failing tests or lack of coverage on a feature branch is acceptable, but the branch should meet these standards before it is merged.)

To check coverage, run `bin/test` and load `htmlcov/index.html` in your browser.

Test methods should set up preconditions for a single action, take that action, and check the results of that single action (generally, separate these three blocks in the test method with blank lines). Multiple asserts in a single test method are acceptable only if they are checking multiple aspects of the result of a single action (even in that case, multiple test methods may be better unless the aspects are closely related). Avoid multi-step tests; they should be broken into separate tests.

Avoid importing the code under test at module level in the test file; instead, import it in helper methods that are called by the tests that use it. This ensures that even broken imports cause only the affected tests to fail, rather than the entire test module.

Prefer helper methods to `TestCase.setUp` for anything beyond the most basic setup (e.g. creating a user for authenticated-view tests); this keeps the setup more explicit in the test, and avoids doing unnecessary setup if not all test methods require exactly the same setup.

Never use external data fixtures for test data; use the object factories in `tests.factories` (available as `self.F` on every `tests.cases.DBTestCase`.) If a large amount of interconnected data is needed, write helper methods. External data fixtures introduce unnecessary dependencies between tests and are difficult to maintain.

Style

A consistent coding style helps make code easier to read and maintain. Many of these rules are a matter of preference and an alternate choice would serve equally well, but follow them anyway for the sake of consistency within this codebase.

If in doubt, follow **PEP 8**, Python's own style guide.

Line length Limit all lines to a maximum of 79 characters.

Docstrings Follow PEP 257. Every module, class, and method should have a docstring. Every docstring should begin with a single concise summary line (that fits within the 79-character limit). If the summary line is the entire docstring, format it like this:

```python
def get_lib_dir():
    """Return the lib directory path."""
```

If there are additional explanatory paragraphs, place both the opening and closing triple-quotes on their own lines. Separate paragraphs with blank lines, and add an additional blank line before the closing triple quote:

```python
def get_lib_dir():
    """
    Return the lib directory path.

    Checks the ``LIB_DIR`` environment variable and the ``lib-dir`` config
    file option before falling back to the default.

    """
```

Docstrings should be formatted using reStructuredText. This means that literals should be enclosed in double back-ticks, and literal blocks indented and opened with a double colon.

Always use triple double-quotes for enclosing docstrings.

Imports Outside of test code, prefer module-level imports to imports within a function or method. If the latter are necessary to avoid circular imports, consider reorganizing the dependency hierarchy of the modules involved to avoid the circular dependency.

Module-level imports should all occur at the top of the module, prior to any other code in the module. The following types of imports should appear in the following order (omitted if not present), each group of imports separated from the next by a single blank line:

1. Python standard library imports.
2. Django core imports.
3. Django contrib imports.
4. Other third-party module imports.
5. Imports from other modules in MozTrap.

Within each group, order imports alphabetically.

For imports from within MozTrap, use explicit relative imports for imports from the same package or the parent package (i.e. where the explicit relative import path begins with one or two dots). For more distant imports, it's usually more readable to give the full absolute path. Thus, for code in `moztrap.view.manage.runs.views`, you could do `from .forms import AddRunForm` and `from ..cases.forms import AddCaseForm`, but it's probably better to do `from moztrap.view.lists import decorators` rather than `from ....lists import decorators`; more than two dots become difficult to distinguish visually.

Never use implicit relative imports; if an import does not begin with a dot, it should be a top-level module. In other words, if `models.py` is a sibling module, always `from . import models`, never just `import models`.

Whitespace Use four-space indents. No tabs.

Strip all trailing whitespace. Configure your editor to show trailing whitespace, or automatically strip it on save. `git diff --check` will also warn about trailing whitespace.

Empty lines consisting of only whitespace are also considered "trailing whitespace". Empty lines should *not* be "indented" with trailing whitespace to match surrounding code indentation.

Separate classes and module-level functions with three blank lines. Separate class methods with two blank lines. Single blank lines may be used within functions and methods to logically group lines of code.

Line continuations Never use backslash line continuations, use Python's implicit line continuations within brackets/braces/parentheses. If necessary, prefer extraneous grouping parentheses to a backslash continuation.

All indents should be exactly four spaces.

The first place to wrap a long line is immediately after the first opening parenthesis, brace or bracket:

```
foo.some_long_method_name(
    arg_one, arg_two, arg_three, keyword="arg")

my_dict = {
    "foo": "bar", "boo": "baz"}

my_list_comprehension = [
    x[0] for x in my_list_of_tuples]
```

If the second line is still too long, each element/argument should be placed on its own line. All lines should include a trailing comma, and the closing brace/paren should go on its own line. (This allows easy rearrangement or addition/removal of items with full-line cut/paste). For example:

```
foo.some_long_method_name(
    foo=foo_arg,
    bar=bar_arg,
    baz=baz_arg,
    something_else="foo",
    )

my_dict = {
    "foo": "bar",
    "boo": "baz",
    "something else": "foo",
    }

my_list_comprehension = [
    x[0] for x in my_list_of_tuples
    if x[1] is not None
    ]
```

One exception to the four-space indents rule is when a line continuation occurs in an `if` test or another block-opening clause. In this case, indent the hanging lines eight spaces to avoid visual confusion between the line continuations and the start of the code block:

```
if (something and
        something_else and
        something_else_again):
    do_something()
```

Comments Code comments should not be used excessively; they require maintenance just as code (an out-of-date comment is often far worse than no comment at all). Comments should add information or context or rationale to the code, not simply restate what the code is doing.

The need for a comment sometimes indicates code that is overly clever or doing something unexpected. Consider whether the code should be expanded for clarity, or the API improved so the behavior is less surprising, before adding a comment.

Use `@@@` in a comment to mark code that requires future attention. This marker should always appear with explanation of why more attention is needed, or what is missing from the current code.

Quotes Always use double-quotes for quoting string literals, unless the quoted string must contain a double-quote character. Quoting such a string with single quotes is preferable to using backslash escapes in the string.

Javascript

Javascript code should pass JSLint.

The *Upgrading* documentation is also applicable to updating your development checkout of MozTrap.

2.3.2 Coding standards

See the *Coding Standards* for help writing code that will maintain a consistent style and quality with the rest of the codebase.

2.3.3 User registration

MozTrap's default settings use Django's "console" email backend to avoid requiring an SMTP server or sending real emails in development/testing mode. So when registering a new user, pay attention to your runserver console; this is where the confirmation email text will appear with the link you need to visit to activate the new account.

2.3.4 Running the tests

To run the tests, after installing all Python requirements into your environment:

```
bin/test
```

To view test coverage data, load `htmlcov/index.html` in your browser after running the tests.

To run just a particular test module, give the dotted path to the module:

```
bin/test tests.model.core.models.test_product
```

Give a dotted path to a package to run all tests within that package, including in submodules:

```
bin/test tests.model.core
```

2.3.5 Compass/Sass

MozTrap's CSS (located in *static/css*) is generated using Sass and the Compass framework, with the Susy grid plugin. Sass source files are located in *sass/*.

The generated CSS is included with MozTrap, so Sass and Compass are not needed to run MozTrap. You only need them if you plan to modify the Sass sources and re-generate the CSS.

To install the necessary Ruby gems for Compass/Sass development, run `bin/install-gems`. Update `requirements/gems.txt` if newer gems should be used.

2.3.6 Loading sample data

A JSON fixture of sample data is provided in `fixtures/sample_data.json`. To load this fixture, run `bin/load-sample-data`.

Warning: Loading the sample data will overwrite existing data in your database. Do not load it if you have data in your database that you care about.

The sample data already includes the *default roles*, so there is no need to run a separate command to create them.

The sample data also includes four users, one for each default role. Their usernames are *tester*, *creator*, *manager*, and *admin*. All of them have the password `testpw`.

Resetting your database

To drop your database and create a fresh one including only the sample data, run these commands:

Note: If your shell user doesn't have the MySQL permissions for the first two commands, you may need to append e.g. *-uroot* to them.

```
mysqladmin drop moztrap
echo "CREATE DATABASE moztrap CHARACTER SET utf8" | mysql
python manage.py syncdb --migrate
bin/load-sample-data
```

If you create a superuser during the course of the `syncdb` command (recommended so that you can access the Django admin), the sample data fixture will not overwrite that superuser.

Regenerating the sample data

The sample data fixture is generated using django-fixture-generator via the code in `moztrap/model/core/fixture_gen.py`, `moztrap/model/environments/fixture_gen.py`, `moztrap/model/tags/fixture_gen.py`, `moztrap/model/library/fixture_gen.py` and `moztrap/model/execution/fixture_gen.py`.

If you've modified one of the above files, you can regenerate the fixture by running `bin/regenerate-sample-data`.

2.3.7 Adding or updating a dependency

Adding a new dependency (or updating an existing one to a newer version) involves a few steps, since the requirements files and both submodules (the requirements tarballs submodule in `requirements/dist` and the *Vendor library* submodule in `requirements/vendor`) must be updated.

Preparing your checkout

By default, the submodules are both checked out via a read-only anonymous URL, so that anyone can check them out. In order to push commits to the submodules, you'll need to switch the push url to use ssh. Make this change as follows:

```
cd requirements/dist
git remote set-url --push origin git@github.com:mozilla/moztrap-reqs

cd ../vendor
git remote set-url --push origin git@github.com:mozilla/moztrap-vendor-lib
```

This assumes that you have permission to push to the primary `moztrap-reqs` and `moztrap-vendor-lib` repositories. If instead you have made your own forks of one or both of these repositories, adjust the above URLs to push to your fork.

Adding the dependency tarball

Assuming the new dependency is a Python package available on PyPI (for the sake of this example we'll assume that we want the 2.1.1 version of the Markdown package), from the root of your MozTrap checkout run this command in order to download the tarball into `requirements/dist`:

```
pip install -d requirements/dist Markdown==2.1.1
```

This should add the `Markdown-2.1.1.tar.gz` file into `requirements/dist`. We want to add this file and commit the change to the submodule. First, though, we need to ensure that we are actually committing on a branch in the submodule, since by default git does not check out submodules on a branch.

In most cases, you can just check out the `master` branch of the submodule and commit there:

```
cd requirements/dist
git checkout master
git add Markdown-2.1.1.tar.gz
# "git rm" the older Markdown tarball, if you're updating
git commit -m "Add Markdown 2.1.1."
git push
```

Note: If you are working on a release branch of MozTrap rather than the master branch, you may find that updating the submodule to `master` updates the version of some dependency to a more recent version, and your branch of MozTrap is not prepared for this dependency update. In that case rather than updating to the submodule's master branch, you should create a new branch of the submodule with a name matching the branch of MozTrap you are working on; replace `git checkout master` in the above with e.g. `git branch 0.8.X`. (If you've already done the `git checkout master`, go back out to the MozTrap repo root and `git submodule update` to get back to the pinned commit of the submodule, then `cd requirements/dist` and `git branch 0.8.X`.) If you create your own branch of the submodule, you may need to also replace `git push` with e.g. `git push -u origin 0.8.X`).

Similarly, if you are working on a feature branch, and your feature branch requires a newer version of a dependency, it is preferable to make a branch of the submodule. The master branch of MozTrap is tied to a specific commit of the submodule, so it won't create an immediate problem if you just push to the submodule's master branch; but if some other feature on the master branch must also update a dependency, there could be a problem if everyone is just pushing to the submodule's master branch. (If you are just adding a dependency, not changing the version of an existing one, this really isn't an issue, as having the extra tarball around won't hurt anything for another branch).

Updating the requirements file

If your added dependency is a pure-Python dependency (no compiled C extensions), add an entry to `requirements/pure.txt` like `Markdown==2.1.1`.

If your added dependency does require compilation, add it to `requirements/compiled.txt` instead.

If you are just updating the version of an existing dependency, find the existing requirement line and change the version.

Updating the vendor library

Note: This step is only necessary for pure-Python dependencies. Compiled dependencies should not be included in the vendor library.

Note: Due to a bug in pip, this step currently must be done within an empty `--no-site-packages` virtualenv. (Virtualenv 1.7+ automatically creates `--no-site-packages` envs by default; with an earlier version you must use the `--no-site-packages` flag).

If you've correctly created and activated a `-no-site-packages` virtualenv, `pip freeze` should show only the `wsgiref` package (which is part of the Python standard library).

Now, from the root of the MozTrap repo, run:

```
bin/generate-vendor-lib
cd requirements/vendor
git status
```

The only changed files shown here should be the new Python files for your added dependency (or, if upgrading a dependency, possibly some added/modified/removed files, but nothing outside the one upgraded package).

If that is the case, commit your changes to the master branch (or the branch you chose earlier) and push using the same steps as shown above for the `requirements/dist` submodule.

Pulling it all together

At this point, if you run `git status` in the root of the MozTrap repo, you should see three modifications: a modification to `requirements/pure.txt` and `(new commits)` in the `requirements/dist` and `requirements/vendor` submodules (or, if you added a compiled module, a modification to `requirements/compiled.txt` and `(new commits)` only in `requirements/dist`).

Add these changes, commit, push, and you're done!

```
git add requirements/
git ci -m "Add Markdown 2.1.1 dependency."
git push
```

2.4 Deployment

Django's built-in `runserver` is not suitable for a production deployment; use a WSGI-compatible webserver such as Apache with mod_wsgi, or gunicorn. A WSGI application callable is provided in `moztrap/deploy/wsgi.py` in the `application` object.

You'll also need to serve the static assets; Apache or nginx can do this.

You'll need a functioning SMTP server for sending user registration confirmation emails; configure the `EMAIL_*` settings and `DEFAULT_FROM_EMAIL` in your `moztrap/settings/local.py` to the appropriate values for your server.

The default local-memory cache backend is not suitable for use with a production (multi-process) webserver; you'll get CSRF errors on login because the CSRF token won't be found in the cache. You need an out-of-process cache backend: memcached or Redis is recommended for production deployment. The Django file or database cache backends may also work for a small deployment that is not performance-sensitive. Configure the `CACHE_BACKENDS` setting in `moztrap/settings/local.py` for the cache backend you want to use.

In addition to the notes here, you should read through all comments in `moztrap/settings/local.sample.py` and make appropriate adjustments to your `moztrap/settings/local.py` before deploying this app into production.

2.4.1 Logins

By default all access to the site requires authentication. If the `ALLOW_ANONYMOUS_ACCESS` setting is set to `True` in `moztrap/settings/local.py`, anonymous users will be able to read-only browse the management and test-results pages (but will not be able to submit test results or modify anything).

By default MozTrap uses BrowserID for all logins, but it also supports conventional username/password logins. To switch to username/password logins, just set `USE_BROWSERID` to `False` in `moztrap/settings/local.py`.

If using BrowserID (the default), you need to make sure that your `SITE_URL` is set correctly in `moztrap/settings/local.py`, or BrowserID logins will not work.

2.4.2 Vendor library

For deployment scenarios where pip-installing dependencies into a Python environment (as `bin/install-reqs` does) is not preferred, a pre-installed vendor library is provided in `requirements/vendor/lib/python`. This library does not include the compiled dependencies listed in `requirements/compiled.txt`; these must be installed separately via e.g. system package managers. The `site.addsitedir` function should be used to add the `requirements/vendor/lib/python` directory to sys.path, to ensure that `.pth` files are processed. A WSGI entry-point script is provided in `moztrap/deploy/vendor_wsgi.py` that makes the necessary `sys.path` adjustments, as well as a version of `manage.py` in `vendor-manage.py`.

If you are using the vendor library and you want to run the MozTrap tests, `bin/test` won't work as it uses `manage.py`. Instead run `python vendor-manage.py test`.

If you need code coverage metrics (and you have the `coverage` module installed; it isn't included in the vendor library as it has a compiled extension), use this:

```
coverage run vendor-manage.py test
coverage html
firefox htmlcov/index.html
```

2.4.3 Security

In a production deployment this app should be served exclusively over HTTPS, since almost all use of the site is authenticated, and serving authenticated pages over HTTP invites session hijacking attacks. The `SESSION_COOKIE_SECURE` setting should be set to `True` in `moztrap/settings/local.py` when the app is being served over HTTPS.

Run `python manage.py checksecure` on your production deployment to check that your security settings are correct.

2.4.4 Static assets

This app uses Django's staticfiles contrib app for collecting static assets from reusable components into a single directory for production serving, and uses django-compressor to compress and minify them. Follow these steps to deploy the static assets into production:

1. Ensure that `COMPRESS_ENABLED` and `COMPRESS_OFFLINE` are both uncommented and set to `True` in `moztrap/settings/local.py`.
2. Run `python manage.py collectstatic` to collect all static assets into the `collected-assets` directory (or whatever `STATIC_ROOT` is set to in `moztrap/settings/local.py`).
3. Run `python manage.py compress` to minify and concatenate static assets.
4. Make the entire resulting contents of `STATIC_ROOT` available over HTTP at the URL `STATIC_URL` is set to.

If deploying to multiple static assets servers, probably steps 1-3 should be run once on a deployment or build server, and then the contents of `STATIC_ROOT` copied to each web server.

2.4.5 Database performance tweak

In order to ensure that all database tables are created with the InnoDB storage engine, MozTrap's default settings file sets the database driver option "init_command" to "SET storage_engine=InnoDB". This causes the SET command to be run on each database connection, which is an unnecessary slowdown once all tables have been created. Thus, on a production server, you should comment this option from your `moztrap/settings/local.py` file's `DATABASES` setting after you've run `python manage.py syncdb --migrate` to create all tables (uncomment it before running `python manage.py syncdb` or `python manage.py migrate` after an update to the MozTrap codebase, or before trying to run the tests).

2.5 User's Guide

This guide provides documentation of MozTrap's underlying concepts and design decisions.

2.5.1 Products

The core object in MozTrap is the **Product**. A Product itself is little more than a name and optional description, but almost every other object in the MozTrap data model relates to a Product either directly or indirectly.

Products have a list of *versions*; every *test run* and *test case version* applies to a particular version of the product.

Product Edit Fields

- **Name** - The name of the Product. (Firefox, Thunderbird, etc)
- **Description** - (optional) A brief description of the product.
- **Version** - Every Product must have at least one Product Version. Many Products will end up with several Product Versions. (1.0, 2.0, 2.5, etc). If this is a web project and you don't want several versions, feel free to call this whatever you like (Production, Current, etc.).
- **Environments** - This is a pre-existing collection of environments called an *Environment Profile*. You can specify this at creation time, or later. Note that the set of environments can be different for different Product Versions because the needs of your product may change over time. When you want to update the list of supported environments, you do this on the Product Version rather than the Product itself.

2.5.2 Product Versions

When a new **Product Version** is created, all test cases for that Product will get a new version to match the new **Product Version**.

For more information on how Test Cases and Product Versions relate while running tests against different builds of a Product, see the *Test Runs* section.

Product versions are automatically ordered according to their *version* number/name. The version is split into dotted segments, and the segments are ordered lexicographically (with implicit left-side zero-padding of numerals to avoid e.g. “2” ordering after “11”). So, for instance, version *1.1* is greater than version *1.0.3*, version *2.0b1* is greater than *2.0a3*, and *3.11.1* is greater than *3.2.0*.

There are some special cases to better support common version-numbering schemes. Strings alphabetically prior to “final” are considered pre-release versions (thus *2.1a*, *2.1alpha*, and *2.1b* are all prior to *2.1*, whereas *2.1g* is considered a post-release patchlevel). The strings “rc”, “pre”, and “preview” are considered equivalent to “c” (thus also pre-release), and the string “dev” orders before “alpha” and “beta” (so *2.1dev* is prior to *2.1a*).

Product versions can also optionally have a *code name* that does not impact their ordering.

Product Version Edit Fields

- **Product** - The Product that this is a version of.
- **Copy Environments From** - (optional) Environments apply to each product version. Each version can have a unique set of environments. But commonly, they are very close, and the set of environments evolves over time. This field allows you to choose which existing product version to copy the environments from. You can then add or remove from the list of environments for this version.
- **Version** - The name of the new version. See *product versions* for more info on how order of versions works.
- **Codename** - (optional) This can be any text and is only used as a reference in the summary list of versions when there is another name for a version. For instance, for Mac OS 10.7, the Codename is *Lion*.

2.5.3 Test Cases and Suites

Test Cases

A **Test Case** is a named set of steps for testing a single feature or characteristic of the system under test. Test cases are associated with a *product*, and can have one version per *product version*. They can be organized via *suites* and/or *tags*, and can have file *attachments*. Preconditions, assumptions, and other preliminary information can be provided in the case’s *description*. A test case can have any number of steps; each step has an *instruction* and an *expected result*.

Case Edit Fields

- **Product** - The product that owns this test case.
- **Version** - The product version of this test case.
- **And Later Versions** - Create a test case version for the specified Product Version as well as a case version for each later Product Version. (e.g.: if Product Versions 3, 4 and 5 exist for this Product, and you have specified Product Version 4, this case will be created for versions 4 and 5)
- **Suite** - (optional) The existing suite to which you want this case to belong. You can also add cases to suites later.
- **Name** - The summary name for the case.

- **Description** - Any description, pre-conditions, links or notes to associate with the case. This field is displayed while running the test. Markdown syntax is supported.
- **Add Tags** - Enter tags to apply to this case. Hit enter after each tag to see the tag chicklet displayed. Auto-completes for existing tags.
- **Add Attachment** - You can attach files to cases that may help running the test. (e.g: images, audio, video, etc.)
- **Instruction / Expected** - The test instruction and corresponding expected result. You can choose to put all instructions / expectations in one step, or break them down to individual steps. When running the test, you will have the option to fail on specific steps, so you may find this a better approach. Markdown syntax is supported.
- **Save** - You can choose to save the case as draft or active. Only active cases can be run in a test run.

Test Suites

A **Test Suite** is a named collection of test cases that can be included in a *test run*.

Suite Edit Fields

- **Product** - The product that owns this test case.
- **Name** - The name of the suite.
- **Description** - Any description for the suite.
- **Available Cases** - Test Cases that have the same Product you selected for this suite. This list is filterable.
- **Included Cases** - Test Cases that are included in the Suite. This list is not filtered.

Tags

A **Tag** can be associated with one or more *test cases* as a way to organize and filter them on any number of axes.

By default, tags are *product*-specific; global tags can also be created and managed via the tag management UI.

Tag Edit Fields

- **Name** - The name of the tag.
- **Product** - (optional) Tags can be specific to a Product, or they can be global. If a tag is Product specific, then cases for other products can't use it. This is useful if you want to separate tags for different products.

Attachments

A *test case* can have any number of file attachments: these will be made available for download by testers when the test case is executed.

2.5.4 Runs and Results

Test Runs

A **Test Run** consists of a set of *test case* versions that can be assigned to a tester for execution (or that a tester can assign to themselves and execute) in a particular *environment* or set of environments.

A test run is for a specific *product* version. It has its own *name*, *status*, *start date*, and *end date*, as well as a list of included *test suites*. A test run must be switched to *active* status before it can be executed by testers.

A Test Run applies to a Product and a Product Version. Usually, a product has had several iterations (or builds) prior to the release of a final Version. Therefore, a Test Run is a single execution pass over a specific iteration of that Product Version. And your product will likely have more than one iteration prior to release of that version. Therefore, you may choose to name your test runs after the build they are testing like Build 23, Build 24, etc. Once your product goes Alpha or Beta, you may choose to name your test runs that way: “Alpha 1, Build 86,” “Alpha 1, Build 87,” etc.

The test case steps executed in test runs may be different for each Product Version, as the Product itself evolves. See *Test Cases* for more info on how test case versions relate to Product Versions.

An active test run can be disabled, which halts all execution of tests in that run until it is made active again.

Cloning Test Runs

If you have a Test Run that you would like to apply to a different Product Version, you must clone the existing Test Run, then edit the new clone while it is still in draft mode. Once your changes are made, you can activate the new run to use it.

Run Edit Fields

- **Product Version** - The product version of this test run. Runs are specific to a version of a product, not just the product in general.
- **Name** - The summary name for the run. When testing a product that has build numbers, you may choose to include the build number in the name to distinguish it from other runs against the same version of the product. Dates in the name are another good way to distinguish runs from one another.
- **Description** - (optional) Any description for the run.
- **Start** - The first date that the run can be executed
- **End** - The date the run expires. A run cannot be executed after its end date.
- **Available Suites** - All the suites that apply to the specified Product Version. This field is filterable.
- **Selected Suites** - The suites from which to gather test cases for this run. When the run is activated, only suites and cases that were active at that time will be included in the run. This field is not filterable.

Test Results

A **Test Result** stores the results of a single execution of one *test case* from a *test run*, in a particular *environment*, by a particular *tester*.

A result has a *status*, which can be any of **assigned** (the test case/environment is assigned to this tester, but hasn’t been run yet), **started** (the tester has started executing the test, but hasn’t yet reported the result), **passed**, **failed**, or **invalidated** (the test case steps were incorrect, did not apply, or the tester couldn’t understand them).

The result also tracks the duration of execution (datetime *started* and *completed*), as well as an optional *comment* from the tester.

A passed/failed/invalidated result can also be recorded for each individual step in the test case, allowing the tester to specify precisely which step(s) failed or were invalid. A failed step can have a *bug URL* associated with it.

2.5.5 Environments

MozTrap allows fine-grained and flexible specification of the environment(s) in which each test should be run.

An **Environment** is a collection of *environment elements* that together define the conditions for a single run of a test. For instance, an environment for testing a web application might consist of a browser, an operating system, and a language; so one environment might be **Firefox 10, OS X, English**, and another **Internet Explorer 9, Windows 7, Spanish**. An **Environment Element** is a single element of a test environment, e.g. **Windows 7** or **Firefox 10**. An **Environment Category** is a category containing several (generally mutually exclusive) elements. For instance, the **Operating System** category might contain the elements **OS X 10.5**, **OS X 10.6**, **Windows Vista**, and **Windows 7**. An **Environment Profile** is a collection of *Environments* that specifies the supported environments for testing a product or type of product. For instance, a **Web Applications** environment profile might contain a set of environments where each one specifies a particular combination of web browser, operating system, and language.

Environment profiles can be named and maintained independently of any specific product; these generic profiles can then be used as the initial profile for a new product. For instance, the generic **Web Applications** profile described above could be used as the initial profile for a new web application product.

Product versions, *runs*, and *test cases* all have their own environment profile; that is, the set of environments relevant for testing that particular product version, test run, or test case. These profiles are *inherited*.

Environment Edit Fields

- **Name** - The name of the *Environment Profile*. This name is what you'll see when selecting environments for a *product version*.
- **Table**
  - **Name** - The name of each *environment category*. Select the environment categories you want to include in your profile. You can create new categories as you need them (see **Add a Category** below)
  - **Elements** - The *environment elements* that exist in this category. You can select **all** elements from a category, or specific ones. You can also create new ones, as you need.
  - **Add a Category** - Click this bar to add a new *environment category*. Just type the new category name in the field and hit enter. You can then add elements to it.
- **save profile** - Clicking this will auto-generate all combinations of the categories and elements you chose above. You will then be taken to a screen where you can pare the list of environments down to only the ones you truly want to have included in the profile. See **Auto-generation** below for more info.

Auto-generation

Given a set of *environment categories* (or subsets of the *elements* from each *category*) MozTrap can auto-generate an environment profile containing every possible combination of one element from each category.

For instance, given the *elements* **Firefox** and **Opera** in the *category* **Browser** and the elements **Windows** and **OS X** in the category **Operating System**, the auto-generated profile would contain the *Environments* **Firefox, Windows**; **Firefox, OS X**; **Opera**, **Windows**; and **Opera, OS X**.

Inheritance

At the highest level, a product version's environment profile describes the full set of environments that the product version supports and should be tested in.

A test run or test case version by default inherits the full environment profile of its product version, but its profile can be narrowed from the product version's profile. For instance, if a particular test case version only applies to the Windows port of the product, all non-Windows environments could be eliminated from that test case's environment profile. Similarly, a test run could be designated as Esperanto-only, and all non-Esperanto environments would be removed from its profile (ok, that's not very likely).

The environment profile of a test case or test run is limited to a subset of the parent product version's profile - it doesn't make sense to write a test case or execute a test run for a product version on environments the product version itself does not support.

When a test case is included in a test run, the resulting "executable case" gets its own environment profile: the intersection of the environment profiles of the test run and the test case. So, for example, if the above Windows-only test case were included in an Esperanto-only test run, that case, as executed in that run, would get an even smaller environment profile containing only Windows Esperanto environments.

Thus, the inheritance tree for environment profiles looks something like a diamond:

```
product-version
  /        \
 run    case-version
  \        /
executable-case-version
```

Cascades Whenever an environment is removed from an object's profile, that removal cascades down to all children of that object. So removing an environment from a product version's profile also automatically removes it from all test runs and test cases associated with that product version.

Adding an environment only cascades in certain situations. Adding an environment to a product version's profile cascades to test runs only if they are still in Draft state; once they are activated, their environment profile can no longer be added to.

Additions to a product version's environment profile cascade only to those test cases whose environment profile is still identical to the product version's environment profile (i.e. test cases that apply to all environments the product supports). Once a test case has been narrowed to a subset of the product version's full environment profile, additions to the product version's profile will have to be manually added to the case's profile if the new environment applies to that case.

Test results, once recorded, are never deleted, even if their corresponding environment is removed from their product version or run's environment profile.

Select Environments

This page allows you to narrow the list of environments for a given object. This can be a *product version*, *test run*, *test suite*, or *test case*. See **Inheritance** and **Cascades** above for a detailed explanation. In this dialog, you can uncheck any environments that you do not want to apply the version/run/suite/case in question. You can also add environments back in that may have been previously removed. Just check or uncheck items to include / exclude them.

2.5.6 Teams

Any *product*, *product version*, or *test run* can optionally have a **Team**, which is just a set of users. Teams are not named or managed as an independent entity; they are simply a set of users associated with a given product, version, or run.

Teams are inherited by default; any product version without its own team explicitly set will inherit its product's team, and any test run without a team set will inherit its product version's team. Unlike *environment inheritance*, there is no subset requirement - a test run can be explicitly assigned any team, even if some members of that team are not part of the product version or product's team.

When a test run is activated, all team members for that test run will automatically be assigned all test cases in that run.

2.5.7 Roles and Permissions

Default roles

Four default roles are created when you run `python manage.py create_default_roles`: *Tester*, *Test Creator*, *Test Manager*, and *Admin*. These roles can be fully customized, and new ones can be created (currently only via the Django admin at `/admin/`).

The default roles have the following permissions:

Tester

- *execute*

Test Creator

All *Tester* permissions, plus:

- *create_cases*
- *manage_suite_cases*

Test Manager

All *Tester* and *Test Creator* permissions, plus:

- *manage_cases*
- *manage_suites*
- *manage_tags*
- *manage_runs*
- *review_results*
- *manage_environments*

Admin

All *Tester*, *Test Creator* and *Test Manager* permissions, plus:

- *manage_products*
- *manage_user*

Permissions

execute

Can run tests and report the results.

create_cases

Can create new test cases and edit them (but not edit test cases created by others). Allows tagging of these test cases with existing tags, but not creation of new tags.

manage_suite_cases

Can add and remove test cases from suites.

manage_cases

Can add, edit, and delete test cases and test case versions.

manage_suites

Can add, edit, and delete test suites.

manage_tags

Can add, edit, and delete tags.

manage_runs

Can add, edit, and delete test runs.

review_results

Can review submitted test results and mark them reviewed.

manage_environments

Can create, edit, and delete environment profiles, categories, elements, and environments.

manage_products

Can create, edit, and delete products and product versions.

manage_user

Can create, edit, and delete users.

2.5.8 Working with Lists

Much of the navigation in MozTrap is done with lists. When managing things like *Test Cases*, it is possible to have a very long list of items.

Filters

You can use filters to narrow down the size of the list you're currently viewing. You can do simple filtering by either clicking certain fields in the list (like tags or Product Version), or by typing them in the filter field.

Another option is to click the "Advanced Filtering" button to show your filter options. Simply click the item value you would like to use for filtering. When it has a check mark next to it, the filter is enabled.

Note: If you have two filters for items of the same type (such as two Tag filters) then the filters are treated as an OR between them, rather than an AND. For instance, if you filter on tags "One" and "Two" the list will reflect items that have EITHER "One" or "Two", not just ones that have both.

Details

Notice there is a triangle on the very left of every list item. Click this triangle to expand and see details about that list item.

2.5.9 Data Import Formats

Note: Imported data should always be UTF-8 encoded.

JSON

JSON is a great way to import more complex sets of *cases* and *suites* for your product. One JSON file will be used per *product version*. Simply use the user interface to create the *Product* and *Version* that applies to the *cases* and *suites* to be imported. Then just import your JSON file to that *product version*.

Simple Example:

```
{
    "suites": [
        {
            "name": "suite name",
            "description": "suite description"
        }
    ],
    "cases": [
        {
            "title": "case title",
            "description": "case description",
            "tags": ["tag1", "tag2", "tag3"],
            "suites": ["suite1 name", "suite2 name", "suite3 name"],
            "created_by": "cdawson@mozilla.com",
            "steps": [
                {
                    "instruction": "instruction text",
                    "expected": "expected text"
                },
                {
                    "instruction": "instruction text",
                    "expected": "expected text"
                }
            ]
        }
    ]
}
```

Both top-level sections ("suites" and "cases") are optional. However, if either section is included, each item requires a "name" field value. Other than that, all fields are optional.

CSV (future)

When importing from a spreadsheet or wiki set of test cases, this may prove a very useful format. This doesn't handle multiple separate steps in test cases. Rather, it presumes all steps are in a single step when imported to MozTrap.

2.5.10 Bulk Test Case Entry Formats

Gherkin-esque

This is one of the test case formats supported in the bulk test case creator.

Format:

```
Test that <test title>
<description text>
When <instruction>
Then <expected result>
```

Example:

```
Test that I can write a test
This test tests that a user can write a test
When I execute my first step instruction
then the expected result is observed
And when I execute mysecond step instruction
Then the second step expected result is observed
```

Markdown (future)

This will be another format for the bulk test case creator.

Example:

```
Test case 1 title here
======================
Description text here

* which can contain bullets
* **with formatting**
   * indentation
   * [and links](www.example.com)

Steps
-----
1. Step 1 action
    * Step 1 Expected Result
2. Step 2 action
    * Step 2 Expected Result

Test case 2 title here
======================
...
```

2.5.11 Frequently Asked Questions

1. **Why don't all of my test cases don't show up when I execute my test run?**

- Your *test cases* or *test suites* may not have been active at the time the *test run* was made active. When a test run is made active, it will take a snapshot of active suites and cases at that time. If cases and suites are made active after that time, they will not show in that test run: only in newly activated test runs. This is because once a test run is activated, it is considered a "unit of work" that won't be altered.
- When you have activated new test cases and/or suites and want a test run to reflect that, you have two options:
    1. clone the existing test run, and activate it.
    2. mark your existing run draft, then active again.

2. **Why don't I see the results I expect when I type in a filter?**

- When you type text into the simple search field, you'll see a drop-down list showing some possible choices. On the right of that list is the field to which that filter will be applied. If you filter for the word "Red" in the *product* field, but there is no product with the word "Red" in it, then you may see a list with no results. When you type your filter word, use the arrow keys to select the field to filter on.

3. **How can I create a test case with no steps?**

- By default, all test cases have steps, and a step has a required field of *instruction*. If you try to save the case when there is an empty instruction, it will say that you must fill out that field. To avoid this, simply click the "X" next to that step, it will be deleted, and you can save your case without steps.

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*